OpenMP Tasks in IBM XL Compilers

Xavier Teruel[1] Priya Unnikrishnan[2] Xavier Martorell[1]

Eduard Ayguadé[1] Raul Silvera[2] Guansong Zhang[2] Ettore Tiotto[2]

[1] Barcelona Supercomputing Center – Universitat Politècnica de Catalunya
C/ Jordi Girona, 31 - 08034 Barcelona (Spain)
{xteruel, xavim, eduard}@ac.upc.edu

[2] IBM Toronto Laboratory
8200 Warden Avenue, Markham, Ontario L6G 1C7 (Canada)
{priyau, rauls, guansong, etiotto}@ca.ibm.com

Abstract

Tasking is the most significant feature included in the new OpenMP 3.0 standard. It was introduced to handle unstructured parallelism and broaden the range of applications that can be parallelized by OpenMP. This paper presents the design and implementation of the task model in the IBM XL parallelizing compilers. The `task` construct is significantly different from other OpenMP constructs. This paper discusses some of the unique challenges in implementing the `task construct` and its associated synchronization constructs in the compiler. We also present a performance evaluation of our implementation on a set of benchmarks and applications. We identify limitations in the current implentation and propose solutions for further improvement.

1 Introduction

OpenMP [2] emerged in the 1990's as a parallel programming model for shared-memory environments. It was centered around expressing structured parallelism (such as parallel loops) in scientific applications. It provides programmers with a simple and flexible interface for developing parallel applications using an incremental approach. The parallelism in the program is expressed using directives that are non-intrusive and preserves the sequential program. The simplicity and ease of use are reasons for its wide acceptance as a *de facto* standard for shared-memory parallel programming.

Modern applications are growing in complexity. Irregular and dynamic structures, such as unbounded loops, recursion and producer/consumer schemes are widely used in applications. It was found that the existing set of OpenMP directives were inadequate to represent and exploit the concurrency available in such applications. The recently published OpenMP 3.0 specification [9] aims to address this shortcoming by introducing a tasking model [3] to support unstructured parallelism. The task model provides two new directives: the `task` directive, which allows the user to identify independent work units, and the `taskwait` directive, a synchronization primitive to synchronize the execution of tasks. The new tasking model introduces a paradigm shift in the OpenMP view from thread-centric to task-centric. The specification also describes the behavioural changes required in existing

constructs such as barriers and locks so that they are meaningful in the presence of tasks. Implicit tasks are also generated at the begining of parallel regions, one for each thread in the team.

Currently, major compiler vendors supporting OpenMP are in the process of implementing the new OpenMP 3.0 standard in their commercial products. The IBM XL compiler family offers C/C++ and Fortran compilers on a variety of Power architecture based systems. The IBM compilers support both automatic parallelization and the OpenMP standard for user-directed shared-memory parallelization. IBM is one of the first compiler vendors to support the OpenMP 3.0 standard with XL C/C++ V10.1 and XL Fortran V12 release for AIX. The XL C/C++ compiler provides full support for the tasking model. The XL Fortran compiler provides partial support for the task model (dynamic arrays as firstprivates on a task are not currently supported).

In this paper, we focus on the design and implementation of the *OpenMP Task model* in the IBM XL compilers. The rest of the paper is organized as follows: Section 2 describes the related work in this area. Sections 3 and 4 presents an implementation overview of compiler transformations and the required runtime support respectively. Section 5 presents an experimental evaluation of our model on several kernel applications. Section 6 some conclusions and finally, Section 7 discusses the future work.

2 Related Work

There have been several proposals for expressing dynamic and irregular parallelism in programming languages. Some of them are based on OpenMP and others on completely new programming model.

Intel *workqueueing* model [15] was the first attempt to add dynamic task generation to OpenMP. This model proposed two new directives as an extension of OpenMP: `taskq` and `task`. The `task` directive defines work units and the `taskq` directive creates a new context for the tasks to execute. *Dynamic Sections* [5] allows dynamic generation of tasks. It was proposed as an extension to the standard OpenMP `sections` construct. Similar to the Intel workqueueing model, the dynamic sections create a new context based on the `sections` construct and allows the definition of independent work units through the `section` directive. This proposal allows nested `section` directives and recursion.

The previous two proposals address the problem of irregular parallelism by extending the current OpenMP worksharing mechanism to include tasks in a limited way. There have also been other proposals that are not based on the OpenMP programming model.The *Cilk* [8] programming language is designed for general-purpose parallel programming as an extension of C. Cilk allows the dynamic generation of tasks using the C extensions. Programmers are required to annotate the program to expose parallelism, leaving the runtime system with the responsibility of scheduling the computation to run efficiently on a given platform.

Cell Superscalar (CellSs) [6] is a programming model which exploits the parallelism of a sequential program on the Cell BE architecture. It is based, as the OpenMP programming model, on a simple annotation of the source code. This information is used at runtime to build a dependence task graph and schedule the execution of function instances as soon as all dependences are satisfied.

Several papers have been published by the OpenMP standards committee during the 2 year period that the tasking model was being designed. The papers discuss the specification [3], some preliminary evalutation of the tasking model [4, 11] and also some possible extensions [12].

The *Nanothread library* [13, 17] and the *Mercurium* [5] research compiler for OpenMP provided a prototype implementation of the new task model which was used by the OpenMP standards committee to evaluate the expressiveness and performance potential of the task model.

3 Compiler Transformations

Parallel compilers generate multi-threaded code corresponding to the OpenMP directives

in the source code. Most compilers use the conventional *outlining* [10] technology to generate multi-threaded code. Outlining is the inverse transformation of inlining. A new subroutine is created for every region associated with an OpenMP directive. Outlining is not the only mechanism available to generate multi-threaded code. For example, Intel generates multi-threaded code using the *multi-entry threading technique* [18, 7] where multiple newly generated regions for the OMP constructs are kept inlined in the same user-defined subroutine.

3.1 Outlining

Traditionally, the OpenMP implementation in the IBM XL family of compilers has used *outlining* to generate multi-threaded code for shared memory parallelization. `parallel` regions and workshare constructs (such as `for`, `sections`, and `single`) are outlined into nested or contained procedures of the original procedure. The advantage of this approach is that any shared automatic variables from the original procedure remain available to the outlined nested procedure through the existing *host association* mechanism. The outlining transformation generates appropriate SMP runtime library calls and passes the address of the outlined procedure to the library. The SMP runtime library ensures that all the threads in the team invoke the outlined routine.

Loop constructs are outlined into parameterized nested procedures so that it can be invoked for different ranges in the iteration space. Figure 1 shows an `omp parallel for` construct and the code after the outlining transformation. `main@OL@1` is a parameterized nested procedure of `main()`. The address of the outlined procedure `main@OL@1` is passed to the SMP runtime library call `xlsmpParallelDoSetup`.

3.2 Challenges

Outlining the parallel code into nested procedures is a well suited approach to synchronous parallelization constructs, such as OpenMP parallel regions and workshare constructs. The outlined procedures for these constructs execute inside the context of the parent procedures.

```
int main()
{
   #pragma omp parallel for
   for(int i=0; i<n; i++)
   a[i] = k;
}

--

long main{}
{
   @_xlsmpEntry0 = xlsmpParSelf();
   if (n > 0)
      xlsmpParallelDoSetup(&main@OL@1...)
   return;
}
void main@OL@1(long @LB58, long @UB59)
{
   @UB0 = @UB59;
   @CIV0 = @LB58;
   do{
      a[][@CIV0] = k;
      @CIV0 = @CIV0 + 1;
   }while (@CIV0 < @UB0);
   return;
}
```

Figure 1: Outlining an OpenMP `parallel for` construct.

OpenMP `task`, on the other hand, can be deferred and executed completely outside the scope of its parent procedures. The procedure containing a `task` may complete and return before the task procedure has started execution; this breaks a fundamental requirement of nested procedures: *The stack frame of the parent procedure must remain active while the nested procedure is in execution.* Implementing asynchronous parallel construct, such as OpenMP `task`'s using *outlining* pose new challenges. Our implementation decision was to use the outlining mechanism for the new `task` construct as well. Reusing the existing OpenMP outlining infrastructure provides many practical engineering advantages over using a completely new implementation.

This section presents some unique challenges that we encountered while implementing the task construct in the compiler.

- *Task function invocation:* On AIX and 64-bit Linux/PPC, what is considered as the function pointer (address of a function's executable code) is actually a pointer to a *function descriptor*. Every function that is externally visible has a function descriptor (see figure 2). The first word contains the address of the function. The second word

```
struct
{
   void *(func_ptr)(); /* func addr */
   void *toc_value;    /* RTOC value */
   void *env;          /* env pointer */
}
```

Figure 2: Function descriptor.

contains the function's TOC pointer. The third word contains an optional environment pointer.

For non-nested functions, the function descriptor is allocated on static storage and all the calls share the same function descriptor. The function descriptors for the nested procedures are allocated on the stack of the parent procedure. Each call has a separate copy of the function descriptor containing the frame pointer required to access the parent's automatic variables. Since a `task` can execute outside the scope of all its ancestors, invoking the address of the outlined subroutine in the SMP runtime library references the function descriptor on the stack of its parent which could have gone out of scope. To enable the invocation of the task function, the entire function descriptor is captured when the `task` is created and stored as part of the `task` data structure in the SMP runtime.

Unlike AIX and Linux 64-bit/PPC, there are no function descriptors on Linux 32-bit/PPC. When the compiler sees an address of a nested function being taken, the compiler inserts several bytes of trampoline code in its place. Any calls via the address of a nested function are actually calls to trampoline code, on the stack. For task procedures on Linux 32-bit/PPC, the trampoline code is captured as part of the `task` data structure as it could go out of scope when the parent/ancestor procedure completes. The trampoline code is later copied back on to the stack and invoked when the task is ready for execution.

- *Allocating firstprivate data:* The private data of any OpenMP construct is allocated on the stack of the outlined routines. So accessing private data is a non issue. For

```
int main()
{
   int x=10;
   #pragma omp parallel firstprivate(x)
   {
      printf("x= %d\n", x);
   }
}

--

void main@OL@1()
{
   x{37} = x{25};
   printf("x= %d/n"),x{37});
   return;
}
long main{23}()
{
   x{25} = 10;
   @_xlsmpEntry0 = _xlsmpParSelf}();
   _xlsmpParRegionSetup(&main@OL@1..)
   rstr = 0;
   return rstr;
}
```

Figure 3: OpenMP firstprivate.

all the synchronous OpenMP constructs, the firstprivate data is allocated on the stack of the new outlined procedure and it is initialized using the stack variables of the parent procedure. In the figure 3, the private x{37} is initialized using the original x{25}. For the synchronous constructs in OpenMP, the outlined procedure is executed as soon as the construct is encountered. So, during the initialization of the firstprivate variables in the child procedure, the stack variables of the parent procedure are guaranteed to be alive.

This mechanism of initializing firstprivates using the stack variables of the parent procedure will not work for the asynchronous `task` constructs. This is because the execution of tasks can be deferred. The stack variables of the parent could have changed by the time the task executes. If the parent completes before the child task, the stack variables of the parent could be out of scope. Figure 4 shows the problem of allocating firstprivates for a `task` on the stack and initializing using the parents variable. It is evident from this figure that the value of i{60} in the parent could have changed when the task is ready to execute.

Handling firstprivate data requires special consideration for the asynchronous `task`

```
int main()
{
   #pragma omp parallel for
   for(int i=0; i<10; i++)
   {
      #pragma omp task firstprivate(i)
      printf("i= %d\n", i);
   }
}

--

long main{}{
   if (n > 0)
      xlsmpParallelDoSetup(&main@OL@2...)
   return;
}
void main@OL@2(unsigned @Lbnd,
               unsigned @UBnd)
{
   @UB = @UBnd;
   i{60} = (long) @LBnd;
   do {
      _xlsmpTaskSetup(&main@OL@1 .. )
      i{60} = (i{60} + 1);
   } while ((unsigned) i{60} < @UB);
   _xlsmpTaskWaitDeep()
   return;
}
void main@OL@1()
{
   i{41} = i{60};
   printf("i= %d/n",i{41});
   return;
}
```

Figure 4: Problem of allocating firstprivates on the stack for `tasks`.

construct. The value of the firstprivate variable for a `task` must be captured when the task is created and not when it starts executing. To accomplish this, firstprivates variables are allocated on the heap and initialized during `task` creation. The firstprivate variables are free'ed when the task completes. To reduce the overhead of heap allocation and freeing, all the firstprivates variables of a `task` are grouped into a single structure. A pointer to this structure is passed to the outlined task routines. All accesses to the firstprivate variable in the outlined task routine are remapped to access the firstprivates indirectly using this address. Hence only one malloc/free pair per task is required to handle firstprivate data. Figure 5 shows the heap allocated firstprivates for the same example in Figure 4.

- *Accessing shared data:* For synchronous OpenMP constructs, shared data is accessed by dereferencing the chain of frame pointers of the parent/ancestor procedures. All the parent procedures in the hierarchy are guaranteed to be alive. This is enabled by the existing *host association* mechanism. There are two scenarios to consider regarding shared data accesses in the asynchronous `task` construct.

  1. *A task shares its private data with its child task.* In this scenario, it is the user's responsibility to insert a task synchronization construct to ensure that the child task completes before the data goes out of scope. Similarly the user must ensure that appropriate task synchronization is in place when automatic variables of a procedure is shared with tasks nested inside the procedure.
  2. *A task accesses data that is shared from a parent synchronous OpenMP constructs.* This data is guaranteed to remain in scope when the `task` executes. A task needs its complete chain of intermediate ancestors in order to access the data of any ancestor. The presence of intermediate nested procedures which can go out of scope breaks the chain of frame pointers required to access the data. The data is in scope but cannot be accessed because of the broken link in the chain.

```
void main@OL@2(unsigned @Lbnd,
               unsigned @UBnd)
{
   @UB = @UBnd;
   i{60} = (long) @LBnd;
   do
   {
      @UNP_PTR0 = malloc(4u);
      (@UNP_PTR  + 0)->@i{52} = i{60};
      _xlsmpTaskSetup_TPO(&main@OL@1,
                          @UNP_PTR);
      i{60} = i{60} + 1
   } while ((unsigned) i{60} < @UB);
   _xlsmpTaskWaitDeep_TPO()
   return;
}
void main@OL@1(char * @UFP0)
{
   printf("i= %d/n",(@UFP0 + 0)->i{52});
   free(@UFP0)
   return;
}
```

Figure 5: Allocating firstprivates on the heap for `tasks`.

```
int main()
{
  int a[10];
  #pragma omp parallel for
  for(int i=0; i<n; i++)
  {
    #pragma omp task
    {
      #pragma omp task
      {
        a[i] = k;
      }
    }
   }
}
```

Figure 6: Task nested inside another task.

```
PROGRAM TEST
INTEGER :: VAL
VAL = 10

!$OMP PARALLEL
  CALL SUB()
!$OMP END PARALLEL

CONTAINS
SUBROUTINE SUB()
!$OMP TASK SHARED(VAL)
  VAL =  1
!$OMP END TASK
END SUBROUTINE
END
```

Figure 7: Task nested inside a FORTRAN contained subroutine.

Figure 6 illustrates this problem when a task is nested inside another task and accesses shared data from an enclosing `parallel` region. The parent task can go out of scope. This breaks the chain of frame pointers required to access the shared array `a` in the child task.

Figure 7 shows a similar problem with tasks nested inside FORTRAN contained subroutine. To access the shared variable `VAL` inside the task, the nested procedure `SUB` must be in scope.

To solve this problem, our implementation *promotes* the nested procedure to the same level of the containing procedure. This mechanism is referred to as *nested procedure promotion*

In this process, we remap all references to shared automatics in the OpenMP `task` to indirect references through an *explicit frame pointer* which is explicitly passed to the outlined procedure.

Effectively, this means that this procedure receives two frame pointers; the system frame pointer is used to access any stack variables on ancestors to the original procedure, and a *parallel frame pointer* used to directly access the stack variables of the original procedures. In cases where multiple tasks are lexically nested inside each other, the innermost tasks will receive multiple frame pointers, one for each containing task that might share any stack variables.

- *Implicit barrier:* A task region binds to the innermost enclosing parallel region. The presence of a task synchronization construct or a barrier guarantees the completion of all the explicit tasks generated in the binding parallel region up to this point.

  The OpenMP specification states that there is an implied barrier at the end of a `parallel` region and worksharing (`for`, `sections`, `single`, `workshare`) constructs without a `nowait` clause. The current IBM implementation invokes the implicit barrier for each thread after the completion of the nested procedure associated with the parallel/worksharing region. An implicit barrier at the end of the nested procedure or immediately after it does not make a difference for synchronous OpenMP constructs. For OpenMP `tasks` however, this can pose a problem when private variables of a parallel/workshare region are shared in explicit tasks generated inside the region. When tasks execute inside the implicit barrier, the private variables of the parent parallel/workshare region have gone out of scope.

  To solve this problem the parent procedures of the tasks need to be alive when the task is executing to ensure correct data access. We address this problem for each of the OpenMP constructs as follows.

  1. For a `parallel` region, the implicit barrier is invoked inside the out-

lined procedure for the parallel region. This will ensure that the parent scope is alive when the generated task executes.

2. The outlined subroutine for the `for` construct is parameterized. The routine can be invoked multiple times by the same thread for different ranges of the iteration space based on the type of schedule. Moving the implicit barrier inside the outlined routine cannot be done in this case. Instead a task synchronization construct, i.e. a `deep taskwait` (refer subsection 4.2), is inserted at the end of the outlined routine. This will ensure that the generated tasks complete before the outlined routine for the `for` construct goes out of scope. This solution has an impact on the performance. A task which is ideally required to complete only by the end of the barrier is now forced to complete at the end of a chunk.

3. Similarly for the `single` construct, a `deep taskwait` is inserted at the end of the outlined routine for the `single` to keep the parent procedure alive for the generated child tasks.

4. For the `sections` construct, each section is outlined into a separate routine. The outlined task routines generated by individual section constructs are promoted up one level (using the *nested procedure promotion* mechanism). The *procedure promotion* mechanism will handle the data access inside the task. The tasks will complete before the end of the implicit barrier of the `sections` worksharing construct.

Tasks generated inside worksharing constructs with the `nowait` clause are promoted up to the level of the worksharing construct. The tasks will complete when the next task synchronization construct or barrier is encountered.

3.3 Host association

Host association is implemented by the compiler backend in a very late transformation. For nested procedures that reference their parent automatic variables, the compiler backend transfers the frame pointer address of the parent procedure as an argument to the child procedure. The child can then access its parents automatics through that pointer. In the case of a hierarchy of nested procedures, a child procedure can dereference a chain of frame pointers to access the local variables of any of its ancestors.

4 Runtime Implementation

The compiler transformed multi-threaded code contains calls to the runtime library. The runtime library is implemented on top of the POSIX threads library and provides support for thread management, synchronization and work scheduling.

The OpenMP 3.0 standard mandates the following constructs.

- `task:` creates an explicit task.
- `taskwait:` a synchronization directive which allows dependences between tasks.

This section presents the implementation of the tasking constructs in the *IBM SMP runtime library* describing the main data structures required for managing tasks, task scheduling and the behavioural changes to existing runtime constructs caused by the introduction of tasks in OpenMP.

4.1 Tasking data structures

4.1.1 Task Pool

All tasks generated in a `parallel` region bind to the parallel region. Each parallel region has an associated task pool (*ready pool*) of tasks ready to execute. The pool is implemented as a queue where newly created tasks are put at the end of the queue and threads pick up tasks from the front of the queue. Tasks are picked up in a pseudo-FIFO order according to certain scheduling restrictions (see Task Scheduling in section 4.3). The maximum size of the

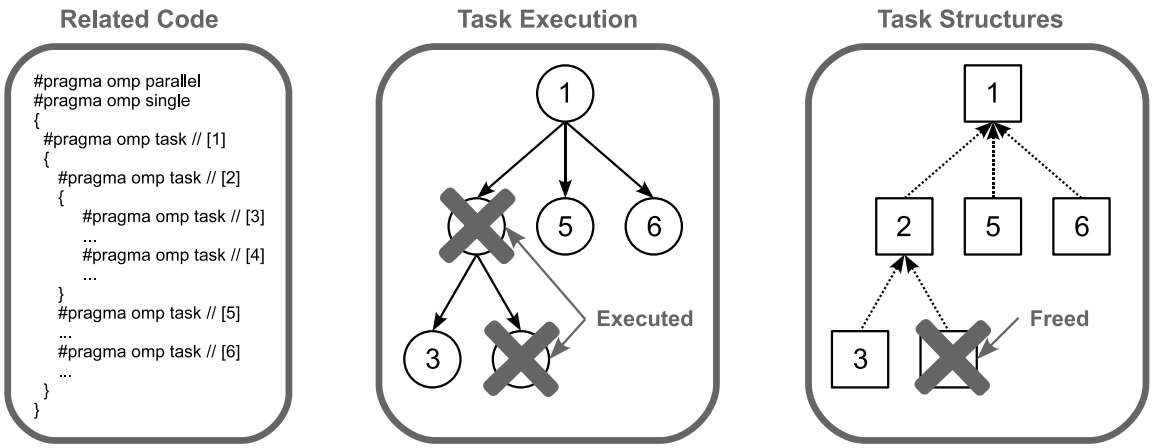

Figure 8: Task structures may persists after a task completes.

task pool may be fixed. Once the pool is full, threads immediately execute the encountered tasks without putting the tasks in the pool. Only tasks which are waiting to be executed and have not been scheduled previously are on the queue. Previously scheduled tasks remain in execution on the *stack frame* of the thread.

The runtime also maintains a pool of free task structures (*free pool*) in order to limit the overhead of task allocation (malloc/free). When a new task is encountered and there are free tasks available in the *free pool*, the runtime picks up a task from this pool instead of allocating a new one. A task can be freed (ie moved to the *free pool*) either by the task itself or one of its children. A task is moved into the *free pool* if, and only if, it has no active children. This means that a task structure may persists even after the code associated with the task completes.

Figure 8 shows an example of this situation. Task-2 is the parent of task-3 and task-4. Task-2 does not have to wait for its children task-3 or task-4 to complete. If task-2 and task-4 complete execution, the task structure for task-2 can persists in memory because task-3 is still in execution. In this example, once task-3 finishes its work it must free the task-2 data structure as it is the last child to finish.

4.1.2 Task

In OpenMP 3.0 tasks can be implicit or explicit. Every thread in a team of a `parallel` region creates an implicit task. Implicit tasks are executed immediately by the encountering thread. The `task` construct creates an explicit task. Only explicit tasks are put on the *ready pool*.

The most relevant features of a task are:

- *A task has code to be executed.* The structured code associated with a task is outlined by the compiler.
- *A task is managed in a task pool.* The task data structure keeps track of the next task in the task pool (*ready pool* or *free pool*).
- *A task must keep information about task hierarchy and task dependences.* A task keeps track of its parent task. It uses counters to track its children/descendant tasks. Operations on these counters are atomic operations.

The task data structure (see Figure 9) keeps track of information about outlined code, task hierarchy, dependences and some task specific characteristics. The structure is initialized by the parent task when the task is generated.

Tasks are serialized in the following situations:

1. When the task is encountered outside a parallel region.

```
typedef struct TASK_STRUCT
{
   void (*sub)();       /* subroutine */
   BOOL untied;         /* flag: tied? */
   long id;             /* task id */
   TASK_STRUCT *next;   /* next on queue */
   TASK_STRUCT *parent; /* parent task */
   long count;          /* dependences */
}TASK;
```

Figure 9: Task data structure.

2. When the binding parallel region has only one thread in the team.

3. When the user program requests a serial task (i.e when the condition on the *if* clause on the `task` construct evaluates to false).

Serialized tasks are not deferred. They are executed immediately by the encountering thread. Although a task is serialized, it must keep track of hierarchy and dependence information. So serialized tasks also have an associated task structure.

4.1.3 Task synchronization

OpenMP 3.0 also offers a mechanism for task synchronization. `taskwait` is an unstructured synchronization construct which must wait for all tasks created in the binding task, up until the `taskwait` construct is encountered. In order to implement this feature, a task needs to know if some of its children have completed before going ahead. Task hierarchy is necessary in order to manage the dependences among tasks. Keeping a counter of active children will allow the task to manage this kind of dependence. A task increments its counter every time it creates a new task and decrements its parent's counter when it completes execution.

4.2 Impact on existing runtime constructs

The paradigm shift in the OpenMP view from thread-centric to task-centric affects the behaviour of the barrier and ownership of locks. This subsection describes the impact of the new OpenMP tasks on these constructs.

4.2.1 Barriers

The OpenMP 3.0 standard requires that all explicit tasks (including children and grandchildren tasks) generated within a parallel region, in the code preceding an explicit or implicit barrier, are guaranteed to be complete on exit from that barrier region. A `taskwait` sychronization construct waits for the immediate decendents of the binding task to complete but a barrier must wait for all descendant tasks. So barriers are also task synchronization points.

A barrier is also a potential *task switching point* (see *Task Scheduling* in section 4.3) where waiting threads can switch to execute tasks of the team when the *ready pool* is not empty. The current barrier implementation must be modified to accomodate this change.

To implement the change in the barrier, a new internal task synchronization construct called `deep taskwait` was introduced. The `deep taskwait` is implemented using a new counter in the task data structure called the *deep* counter. The *deep* counter is incremented when a child task is created. Completed tasks decrement the *deep* counter of its parent task only when all its own children have completed execution. Once a child task has decremented its parent's *deep* counter and sees that the parent has no other pending children, it must go up the chain of ancestor tasks and decrement their deep counters appropriately.

Using the `deep taskwait` construct, the new barrier can be easily implemented in two phases:

1. The implicit task of a thread must wait for all descendants. (ie `deep taskwait`).

2. Regular thread barrier implementation.

4.2.2 Locks

Ownership of locks has changed from OpenMP 2.5 to OpenMP 3.0. In OpenMP 2.5, locks are owned by threads; so a lock released by the *omp_unset_lock* routine must be owned by the same thread executing the routine. In OpenMP 3.0, on the other hand, locks are owned by task regions; so a lock released by the *omp_unset_lock* routine in a task region must be owned by the same task region.

Lock ownership was previously enforced using the unique *thread_id* associated with each thread in the team. To express the lock ownership in terms of tasks, the runtime assigns a unique *task_id* for every task created both explicit or implicit. The runtime also reserves *task_id* 0 (zero) for tasks that are created outside a parallel region.

4.3 Task Scheduling

Task switching is the ability of a thread to suspend the execution of the current task and execute a different task. A thread may switch from one task at a *suspend point* to a different task at a *resume point*. For a tied task, a thread can *task switch* when it encounters a task, when it is waiting at a taskwait or at an OpenMP barrier (implicit or explicit). For an untied task the task switching can happen anywhere in the code. These points are called *Task Switching Points* (TSP).

The OpenMP 3.0 specification mandates the following scheduling restrictions:

1. An explicit task whose construct contained an `if` clause whose scalar-expression evaluated to false is executed immediately after generation of the task.

2. Other scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant of every task in the set.

When an `if` clause is present on a task construct and the value of the scalar expression evaluates to false, the thread suspends the current task region and begins executing the encountered task immediately. The suspended task region is not resumed until the encountered task is completed.

Tied tasks are scheduled if, and only if, we can assure that it is a descendant of every task on the set of tied tasks currently executed by the thread. As the IBM runtime leaves all current tied tasks of the thread on its *stack frame* and the tied task on the top of the *stack frame* is a descendant of the previous ones (due the scheduling restrictions) we only have to check if the new tied task is a descendant of the top most tied task on the *stack frame*. When the TSP is in a barrier, the tasks are scheduled in a pure FIFO order with no restrictions.

Kernel App.	*Seq. T*	*Par. T*	*Overhead*
SparseLU	45.01 s.	45.09 s.	+0.08 s.
Alignment	93.14 s.	91.19 s.	-1.95 s.
Multisort	32.16 s.	32.18 s.	+0.02 s.
Queens	94.63 s.	94.63 s.	+0.00 s.

Table 1: Benchmark execution time and overhead (seconds).

5 Experimental Results

For a preliminary performance evaluation of our implementation of the task model, we have used several kernel applications such as *alignment*, *queens*, *sparseLU* and *multisort*. These benchmarks have been used previously to evaluate the tasking model in the *Nanos environment* [13]. This allows us to compare the performance of the two implementations in terms of the speed-up achieved.

Alignment kernel computes the alignment of protein sequences. Based on the Smith-Waterman [16] algorithm it compares segments of all possible lengths and determines the alignment and similarity of protein sequences. The benchmark receives as a parameter an input file with all protein sequences. The output is the best score for each pair of them. In our experiments we use a sample file of `100` protein sequences.

N-Queens problem tries to find a placement for `N` queens on an `NxN` chessboard such that none of the queens attacks any other. The algorithm computes all solutions of the N-Queens problem using a backtracking search algorithm. In our experiments we use a chessboard size of `14x14`.

The *sparseLU* benchmark computes an LU matrix factorization over sparse matrices. Due to the sparseness of the matrix, work-sharing solutions have to deal with a lot of load im-

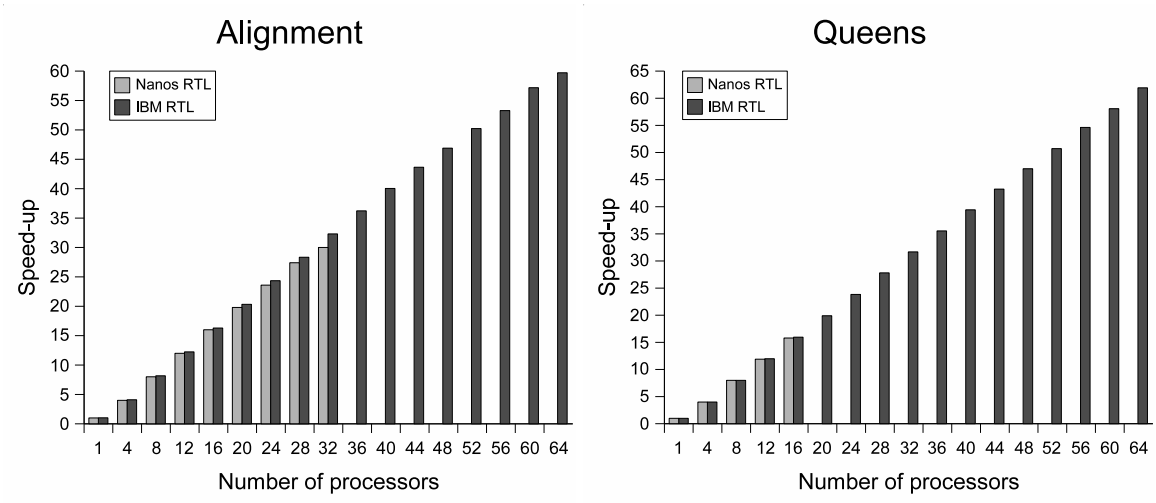

Figure 10: Alignment and Queens results (Speed-up).

balance. In our experiments, the matrix size is set at `5000x5000` and submatrix sizes are set at `100x100` float elements. It is a useful benchmark for testing OpenMP tasks as the matrix size is proportional to the number of created tasks and submatrix size determines the task granularity.

Multisort is a variation of the ordinary mergesort algorithm. It uses a parallel divide and conquer mergesort [1] and a serial quicksort [14] when the array is too small. In our experiments we use array size set to `128 Mb`.

All the experiments were performed on an IBM Power5 machine with 64 SMT processors running at 1656 MHz, 512 GB memory and executing AIX 5.3. Benchmarks were compiled for 64 bits binary code using IBM XL C/C++ Enterprise Edition for AIX, V10.1 with `-O3` and `-q64` flags for both the sequential and parallel versions. In addition, the parallel version uses an extra `-qsmp=omp` flag. At runtime, thread binding is enabled to bind to every other processor using the environment variable `XLSMPOPTS=startproc=0:stride=2` and threads are forced to busy-wait by setting `XLSMPOPTS=spins=0:yields=0`.

Table 1 shows the overhead of using the tasking implementation in the runtime library. The table compares the sequential execution time of the benchmark with the execution time when OpenMP tasks are used with 1 thread. Results show that the overhead of OpenMP tasking model is very small and does not impact the execution time of this set of kernel application.

Figures 10 and 11 present the speed-up obtained for all benchmarks using sequential execution time as baseline. In the *alignment* and *queens* benchmarks we observed an almost linear speed-up. *SparseLU* also shows a linear speed-up upto 44 threads before reaching *saturation*. The *multisort* kernel shows a lower and more irregular speed-up when compared to the Nanos RTL.

The task generation pattern in *multisort* seems to be the likely cause for this irregularity. Figure 12 shows the task generation for the *multisort* kernel. *Multisort* uses a recursive algorithm for splitting the work among descendants tasks and merging results after the work is done. Each task executes two task synchronization primitives.

The current IBM runtime implements untied tasks as tied tasks. Once task generation reaches the leaf nodes on the recursive scheme all tasks are already bound to a specific thread (i.e. they become *tied*) and are available on the *stack frame* of the thread. This prevents threads from executing tasks bound to other threads causing some unbalanced execution. This seems to be the main cause of the irregular behaviour in *multisort* but a more detailed analysis is required on this issue. The

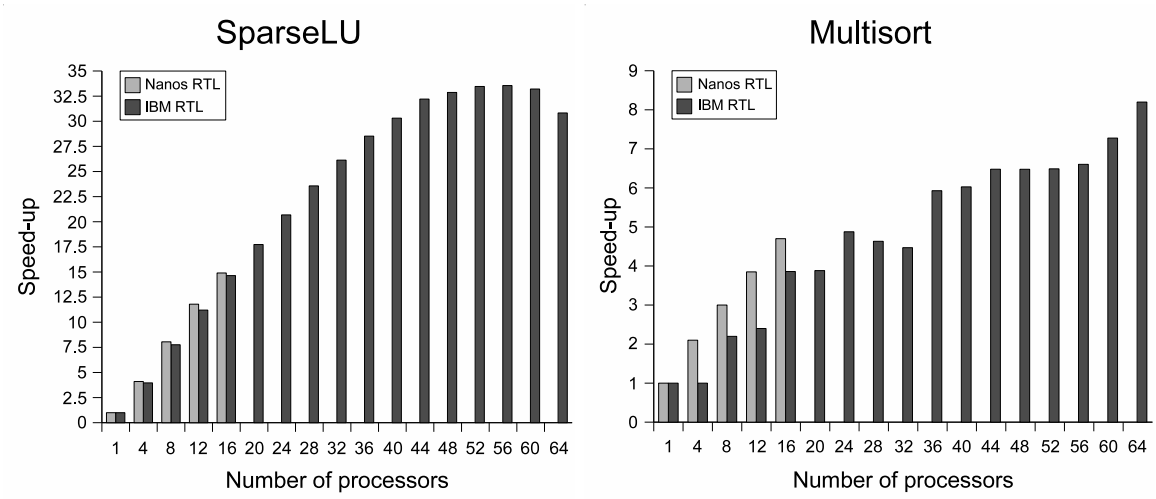

Figure 11: SparseLU and Multisort results (Speed-up).

Nanos RTL allows an untied task to be executed by any available thread.

6 Conclusions

In this paper we have presented the implementation of OpenMP Tasks in the IBM XL compilers. The paper discusses some of the implementation challenges posed by the asyschronous nature of tasks and how they were addressed. We also presented details of task management, the scheduling mechanism and synchronization algorithms implemented in the SMP runtime library. The new OpenMP features were incorporated within the compiler and runtime without any significant problems or impact on the performance of existing OpenMP constructs.

We also presented some preliminary performance results using our initial implementation. Although we have found some irregular behaviour with some application kernels we can conclude that, in general, applications could take advantage of tasking model.

7 Future Work

The current implementation does not support dynamic arrays in FORTRAN and variable lengh arrays (VLA) in C as firstprivates on a

```
void sort (ELM *low, ELM *tmp, long size)
{
   .
   .
   .
   /* split array in A,B,C and D */
   #pragma omp task untied
   sort(A, tA, quart);
   #pragma omp task untied
   sort(B, tB, quart);
   #pragma omp task untied
   sort(C, tC, quart);
   #pragma omp task untied
   sort(D, tD, size - 3 * quart);

   #pragma omp taskwait

   /* merge (AB) */
   #pragma omp task untied
   merge(A, A+quart-1, B, B+quart-1, tA);
   /* merge (CD) */
   #pragma omp task untied
   merge(C, C+quart-1, D, low+size-1,tC);

   #pragma omp taskwait

   /* merge (AB)(CD) */
   merge(tA, tC-1, tC, tA+size-1, A);
   .
   .
   .
}
```

Figure 12: Merge sort task generation.

task. This was mainly due to the lack of time. Another area that requires careful study is the interaction of tasking with C++ exception handling.

The runtime implementation currently supports *tied* tasks only. *Untied* tasks are implemented as *tied* tasks. Part of the future work is to implement *untied* tasks and study the advantages (if any) of using *untied* tasks. Despite the lack of performance tuning, we have seen performance gains using the current implementation of OpenMP tasks. There is still a lot of scope for improvement. We would like to experiment with faster mechanism of managing the tasks in the pool. Possibly have multiple pools instead of one central pool and experiment different scheduling mechanisms.

In the near future, our goal is to do a detailed performance evaluation and comparison with other OpenMP Task implementations on a wider set of applications.

Acknowledgements

This work has been supported by the IBM Center for Advanced Studies (CAS), Toronto, the BSC-IBM MareIncognito project, the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC network of Excellence (IST-004408), and the Spanish Ministry of Education (contract no. TIN2007-60625).

About the Authors

Xavier Teruel received his B.Sc. (2003) and his M.Sc. (2006) in Computer Science at *Universitat Politècnica de Catalunya* (UPC). Currently he is a Ph.D. student at *Computer Architecture Department* from the same university. Xavier's research interests include shared memory environments and parallel programming models. Since 2006 he is working as a researcher within the Parallel Programming models team in the Computer Sciences Department at *Barcelona Supercomputing Center* (BSC).

Priya Unnikrishnan is a Staff Software Engineer in the Compiler Group at the *IBM Toronto Laboratory* since 2003. She works on the IBM XL compilers focussing on OpenMP and automatic parallelization. Her areas of interest are parallel computing, parallelizing compilers, tools and multi-core architectures. She represents IBM at the OpenMP Language Committee. She received an MS degree in Computer Science and Engineering from *The Pennsylvania State University* in 2002.

Dr. Xavier Martorell received the M.Sc. and Ph.D. degrees in computer science from the *Universitat Politècnica de Catalunya* (UPC) in 1991 and 1999, respectively. Since 1992 he has lectured on operating systems, parallel runtime systems and OS administration. He has been an associate professor in the *Computer Architecture Department* at UPC since 2001. His research interests cover the areas of operating systems, runtime systems, compilers and applications for high-performance multiprocessor systems. He spent one year working with the BG/L team in the *IBM Watson Research Center*. He is currently the Manager of the Parallel Programming Models team in the Computer Sciences Department at the *Barcelona Supercomputing Center* (BSC).

Dr. Eduard Ayguadé received the Engineering degree in Telecommunications in 1986 and the Ph.D. degree in Computer Science in 1989, both from the *Universitat Politècnica de Catalunya* (UPC). Since 1987 he has been lecturing on computer organization and architecture and parallel programming models. Currently, and since 1997, he is full professor of the Computer Architecture Department at UPC. He is currently associate director for research on Computer Sciences at the *Barcelona Supercomputing Center* (BSC). His research interests cover the areas of multicore architectures, and programming models and compilers for high-performance architectures.

Raul Silvera is a senior developer at the *IBM Toronto Lab.* After joining IBM in 1997, he has focused on compilation technology and made contributions in the areas of loop transformation, locality analysis, profile-directed optimization, automatic and user-directed parallelization, interprocedural optimizations and others. He is currently the technical lead for TPO, the mid-level optimizer used in the IBM XL Family of compilers. He has participated on the development of the OpenMP standard,

and also on the ISO C++ Language standard committee.

Dr. Guansong Zhang is a Staff Software Engineer in the Compiler Group at the *IBM Toronto Laboratory* since 1999 in charge of OpenMP implementation and performance improvement for Fortran and C/C++ on PowerPC and Cell architecture and Performance related compiler optimization techniques, including array data flow analysis, loop optimization and auto parallelization. Previously, he was a research scientist at NPAC center at *Syracuse University*. He received his Ph.D. from the *Harbin Institute of Technology* in 1995.

Ettore Tiotto graduated *Summa Cum Laude* in Physics from the *University of Torino*, Italy, in 1996. Since joining *IBM Toronto Laboratory* in 1999 he has worked on numerous releases of the industry leading XL C/C++ compiler for AIX, Linux, and Mac OS X. In his earlier career Mr. Tiotto focused on the development of language extensions for the C compiler, with particular focus on ISO C99 conformance, Altivec extensions, GNU inline assembly and other GNU C language extensions. In 2004 he designed and developed a technology preview of an Objective-C compiler for the Mac OS X platform. Since 2006 Mr. Tiotto has participated in ongoing research and development of a prototype UPC compiler (recipient of the High Performance Computing Challenge Class 2 Award at the Super Computing conference in 2006 and 2007).

References

[1] S.G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.*, 36(11):1367–1369, 1987.

[2] Architecture Review Board. *OpenMP Fortran Application Program Interface v 1.0*, October 1997.

[3] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A proposal for task parallelism in OpenMP. In *3rd International Workshop on OpenMP (IWOMP)*, June 2007.

[4] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An experimental evaluation of the new openmp tasking model. In *Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, October 2007.

[5] J. Balart, A. Duran, M. González, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: a Research Compiler for OpenMP. In *6th European Workshop on OpenMP (EWOMP'04)*, Stockholm, Sweden, October 2004.

[6] P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[7] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. In *Intel Technology Journal Q1*, pages 1–9, March 2001.

[8] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[9] Architecture Review Board. *OpenMP Application Program Interface v 3.0*, May 2008.

[10] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar. Automatic Parallelization for Symmetric Shared-Memory Multiprocessors. In *Proceedings of CASCON*, pages 76–89, November 1996.

[11] A. Duran, J. Corbalan, and E. Ayguadé. Evaluation of openmp task scheduling strategies. In *IWOMP '08: International Workshop on OpenMP (IWOMP)*, pages 100–110, May 2008.

[12] A. Duran, J.M. Perez, E. Ayguadé, R.M. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In *IWOMP '08: International Workshop on OpenMP (IWOMP)*, pages 111–122, May 2008.

[13] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé. Nano-Threads Library Design, Implementation and Evaluation. Technical Report UPC-DAC-1995-33, DAC/UPC, September 1995.

[14] R. Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, 1978.

[15] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in openmp. *Concurrency - Practice and Experience*, 12(12):1219–1239, 2000.

[16] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[17] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé. Support for openmp tasks in nanos v4. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 256–259, New York, NY, USA, 2007. ACM.

[18] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su. Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. In *Intel Technology Journal 6*, pages 36–46, 2002.